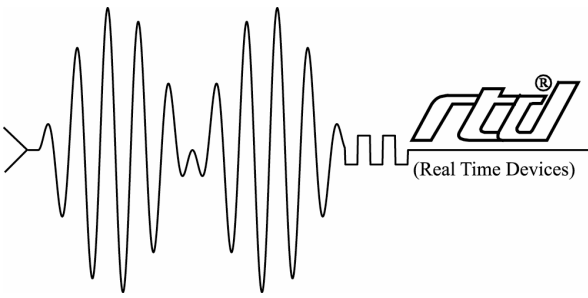
DM5812/DM6812 Driver for Linux

Driver Version 2.0.x User's Manual



RTD Embedded Technologies, Inc.
"Accessing the Analog World"®

SWM-640010020
rev B

ISO9001 and AS9100 Certified

RTD Embedded Technologies, Inc.
103 Innovation Boulevard
State College, PA 16803-0906

Phone: +1-814-234-8087
FAX: +1-814-234-5218

E-mail
sales@rtd.com
techsupport@rtd.com

web site
http://www.rtd.com

Revision History

02/07/2005	Revision A issued Documented for ISO9000
09/29/2005	Revision B issued Deleted discussion in “Interrupt Performance” section, replacing it with reference to application note SWM-640000021

DM5812/DM6812 Driver for Linux
Published by:

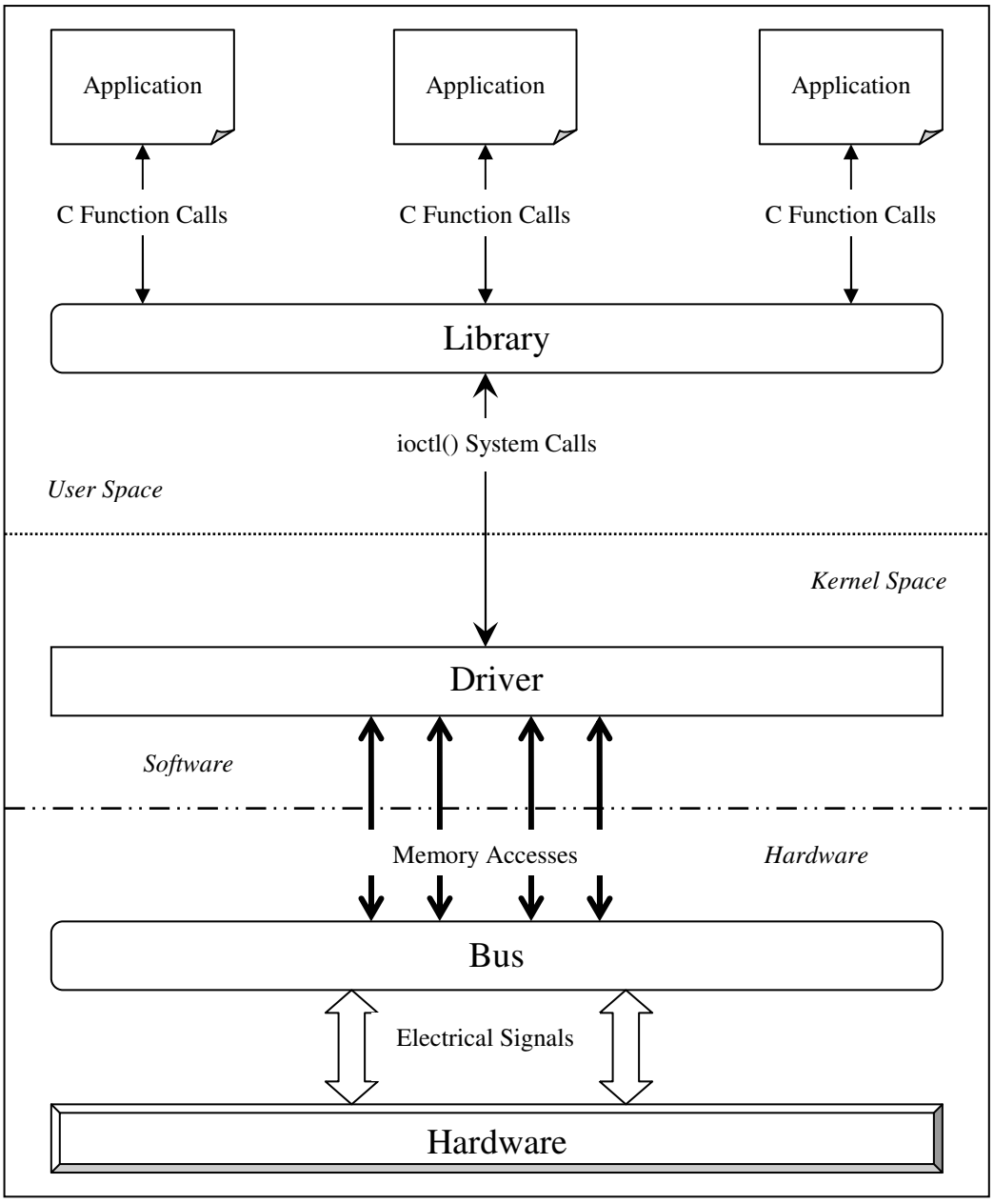
RTD Embedded Technologies, Inc.
103 Innovation Boulevard
State College, PA 16803-0906

Table of Contents

Introduction

This document targets anyone wishing to write Linux applications for an RTD DM5812 or DM6812 dataModule. It provides information on building the software and about the Application Programming Interface used to communicate with the hardware and driver. Each high-level library function is described as well as any low-level ioctl() system call interface it may make use of.

The diagram below 1) provides a general overview of what hardware and software entities are involved in device access, 2) shows which units communicate with each other, and 3) illustrates the methods used to transfer data and control information.



Notational Conventions

RTD Linux drivers are assigned version numbers. These version numbers take the form "**A**.**B**.**C**" where:

* **A** is the major release number. This will be incremented whenever major changes are made to the software. Changing the major release number requires updating the software manual.
* **B** is the minor release number. This will be incremented whenever minor, yet significant, changes are made to the software. Changing the minor release number requires updating the software manual.
* **C** is the patch level number. This will be incremented whenever very minor changes are made to the software. Changing the patch level number does not require updating the software manual.

In this document, you will see driver version numbers with a letter in them. For example, 2.0.x indicates that the topic being discussed is applicable to driver versions with a major release number of 2, a minor release number of 0, and any patch level number.

Occasionally you will notice text placed within the < and > characters, for example <installation path>. This indicates that the text represents something which depends upon choices you have made or upon your specific system configuration.

Installation Instructions

Extracting the Software

All software comes packaged in a gzip'd tar file named dm6812_Linux_v2.0.x.tar.gz. First, decide where you would like to place the software. Next, change your current directory to the directory in which you have chosen to install the software by issuing the command "cd <installation path>". Then, extract the software by issuing the "tar -xvzf <path to tar file>/dm6812_Linux_v2.0.x.tar.gz" command; this will create a directory dm6812_Linux_v2.0.x/ that contains all files comprising the software package.

Contents of Installation Directory

Once the tar file is extracted, you should see the following files and directories within dm6812_Linux_v2.0.x/:

driver/
examples/
include/
lib/
CHANGES.TXT
LICENSE.TXT
README.TXT

The file CHANGES.TXT describes the changes made to the software for this release, as well as for previous releases. The file LICENSE.TXT provides details about the RTD end user license agreement which must be agreed to and accepted before using this software. The file README.TXT contains a general overview of the software and contact information should you experience problems, have questions, or need information. The directory driver/ contains the source code and Makefile for the drivers. The directory examples/ holds the source code and Makefile for the example programs. The directory include/ contains all header files used by the driver, example programs, library, and your application programs. Library source code and Makefile reside in the directory lib/.

Building the Driver

Driver source code uses files located in the kernel source tree. Therefore, you must have the full kernel source tree available in order to build the driver. The kernel source tree consumes a lot of disk space, on the order of 100 to 200 megabytes. Because production systems rarely contain this much disk space, you will probably use a development machine to compile the driver source code. The development system, which provides a full compilation environment, must be running the exact same version of the kernel as your production machine(s); otherwise the kernel module may not load or may load improperly. After the code is built, you can then move the resulting object files, libraries, and executables to the production system(s).

Building the driver consists of several steps: 1) compiling the source code, 2) loading the resulting kernel module into the kernel, and 3) creating hardware device files in the /dev directory. To perform any of the above steps, you must change your current directory to driver/. The file Makefile contains rules to assist you.

To compile the source code, issue the command “make”. The GNU C compiler gcc is used to build the driver code. This will create the driver object file, which is named rtd-dm6812.o on 2.4 kernels and rtd-dm6812.ko on 2.6 kernels.

Before the driver can be used, it must be loaded into the currently running kernel. Using the command “make insmod” will load the DM6812 driver into the kernel. This target assumes that:

* A single DM6812 is installed.
* The board's base I/O address is set to the factory default of 0x300.
* The DM6812 is jumpered to use IRQ 5.

If the previous assumptions do not match your hardware setup, you will need to edit the Makefile and change this rule to reflect your board configuration or manually issue an appropriate insmod command.

For the 2.4 kernel, when you load the kernel driver the message

"Warning: loading rtd-dm6812.o will taint the kernel: non-GPL license - Proprietary"

will be printed on your screen. You can safely ignore this message since it pertains to GNU General Public License (GPL) licensing issues rather than to driver operation.

For the 2.6 kernel, when you load the kernel driver, no warnings will appear on your screen. However, the warning

"module license 'Proprietary' taints kernel."

will be written to the system log when the module is loaded. You can safely ignore this message since it pertains to GNU General Public License (GPL) licensing issues rather than to driver operation.

The final step is to create /dev entries for the hardware. Versions of the driver prior to 2.0.0 always assumed a character device major number of 254 when registering the boards and creating the /dev entries. Instead, the driver now asks the kernel to dynamically assign a major number. Since this major number may change each time you load the driver, the best way to create the device files is to use the command "make devices"; this generates four files in /dev named rtd-dm6812-0 through rtd-dm6812-3.

Be aware that driver/Makefile uses the "uname -r" command to determine which kernel version it is running on. It does this to set up separate make rules and variables for the 2.4 and 2.6 kernels, which allows one set of targets to work on both kernels. Compiling the driver on a development machine which does not run the same kernel version as the production machine that will host your application almost certainly invites trouble.

If you ever need to unload the driver from the kernel, you can use the command "make rmmod".

Building the Library

The example programs and your application use the DM6812 library, so it must be built before any of these can be compiled. To build the library, change your current directory to lib/ and issue the command “make”. The GNU C++ compiler g++ is used to compile the library source code. To prevent compatibility problems, any source code which makes use of library functions should also be built with g++.

The DM6812 library is statically linked and is created in the file librtd-dm6812.a.

Building the Example Programs

The example programs may be compiled by changing your current directory to examples/ and issuing the command “make”, which builds all the example programs. If you wish to compile a subset of example programs, there are targets in Makefile to do so. For example, the command “make digital-io interrupt-wait” will compile and link the source files digital-io.cpp and interrupt-wait.cpp. The GNU C++ compiler g++ is used to compile the example program code.

Interrupt Performance

Performance Issues

Many factors exist outside of the driver software that may impact interrupt performance and throughput. For a discussion of these issues, please see the Application Note SWM-640000021 (Linux Interrupt Performance) on our web site.

Rewriting the Interrupt Handler

Suppose that you have the following requirements: 1) the only interrupt being used is the event mode digital interrupt on port 0, 2) whenever a digital interrupt occurs, the value on digital I/O port 2 should be read, and 3) on digital interrupt occurrence, the value 0x01 should be written to digital I/O port 5. Furthermore, assume the interrupts occur fast enough that dealing with them in user space is unreliable.

Because the interrupt handler provided with the driver offers generic services suitable for a wide range of purposes, it does not provide the functionality indicated above. Therefore to meet your requirements, you must rewrite the interrupt handler. A detailed description of writing interrupt handlers lies beyond the scope of this document. What follows is pseudo code for an interrupt handler which implements the desired behavior.

```
interrupt_handler() {

    /*
     * Determine interrupt status
     */

    read IRQ Status Register at offset 0x11;
    if bit 0 is not set in register
    then

        /*
         * Spurious interrupt
         */

        exit;
    end if

    /*
     * Acknowledge the interrupt.  This consists of two steps: 1) writing the
     * appropriate value to the Port 0/1 Program Digital Mode Register to select
     * clear mode access to Port 0 Clear IRQ Register, and 2) reading Port 0 Clear
     * IRQ Register to acknowledge and clear the interrupt.
     */

    write to Port 0/1 Program Digital Mode Register at offset 0x03;
    read Port 0 Clear IRQ Register at offset 0x02;

    /*
     * Read value on digital I/O port 2
     */

    read Digital I/O Port 2 Register at offset 0x04;

    /*
     * Write 0x01 to digital I/O port 5
     */

    write 0x01 to Digital I/O port 5 Register at offset 0x09;
}
```

Using the API Functions

DM6812 hardware and the associated driver functionality can be accessed through the library API (Application Programming Interface) functions. Applications wishing to use library functions must include the include/dm6812_library.h header file and be statically linked with the lib/librtd-dm6812.a library file.

Because of changes made in driver version 2.0.0, existing source code which uses the library will not compile. Some of the areas requiring attention on your part are:

* All header files have been renamed. Users upgrading from a previous driver version will need to modify source code to include the appropriate header files.
* All header files have been relocated to include/. Be sure to update any files which contain hardcoded header file paths.
* 6812 has been appended to all library function names with the exception of class constructors and destructors.
* DIO has been prepended to all library function names that pertain to digital I/O.
* Library functions which accepted a digital I/O chip number now accept a digital I/O port number.
* The driver and library no longer support interrupt notification via signals. The new notification paradigm provides a function which blocks in the kernel until an interrupt occurs.
* The driver now acknowledges all interrupts in the interrupt handler. Therefore, applications can no longer obtain direct board status. However, the driver caches the IRQ Status Register value and a program may obtain this instead.
* All classes have been coalesced into a single class DM6812Device.
* The DM6812 /dev entry file names have changed. For example, the device file previously named /dev/rtd/dm6812hr/device0 is now called /dev/rtd-dm6812-0.
* Obsolete functions have been deleted. You will need to identify these functions and replace them with alternate functionality if appropriate.
* Some functions require additional parameters to pass non-status information back to the caller, thus allowing the function return value to be an error indication.

The following function reference provides for each library routine a prototype, description, explanation of parameters, and return value or error code. By looking at a function's entry, you should gain an idea of: 1) why it would be used, 2) what it does, 3) what information is passed into it, 4) what information it passes back, 5) how to interpret error conditions that may arise, and 6) the ioctl() system call interface if the function makes use of a single ioctl() call.

Note that errno codes other than the ones indicated in the following pages may be set by the library functions. Please see the ioctl(2) man page for more information.

Function Reference

API Function Groups

Digital I/O

DIOClearChip6812
DIOClearStrobe6812
DIOEnableIrq6812
DIOIsStrobe6812
DIORead6812
DIOReadCompare6812
DIOSelectClock6812
DIOSetBitDirection6812
DIOSetCompareValue6812
DIOSetIrqMode6812
DIOSetMaskValue6812
DIOSetPortDirection6812
DIOWrite6812

General

CloseBoard6812
DM6812Device
~DM6812Device
GetDriverVersion6812
OpenBoard6812
ReadByte6812
WriteByte6812

Interrupt Control and Status

GetIntStatus6812
LoadIRQRegister6812
WaitForInterrupt6812

Timer/Counter Control and Status

ClockDivisor6812
ClockMode6812
ReadTimerCounter6812
SetUserClock6812

Alphabetical Function Listing

ClockDivisor6812

bool ClockDivisor6812(u_int8_t Timer, u_int16_t Divisor);

Description:

Set the divisor for the given 8254 timer/counter.

NOTE: Before calling this function, you must ensure that the indicated timer/counter is set to be programmed least significant byte first then most significant byte.

Parameters:

Timer: The timer to operate on. Valid values are:
- 0 Timer/counter 0
- 1 Timer/counter 1
- 2 Timer/counter 2

Divisor: Counter divisor. Valid values are 0 through 65535.

Return Value:

true: Success.

false: Failure with errno set as follows:

EINVAL Timer is not valid.

Please see the description of the internal function outb() for information on other possible values errno may have in this case.

IOCTL Interface:

This function makes use of several ioctl() requests.

ClockMode6812

bool ClockMode6812(u_int8_t Timer, u_int8_t Mode);

Description:

Set the mode for the given 8254 timer/counter.

NOTE: This function puts the indicated timer/counter into binary mode.

NOTE: This function sets the timer/counter to read/load least significant byte first then most significant byte.

Parameters:

Timer: The timer to operate on. Valid values are:

- 0 Timer/counter 0
- 1 Timer/counter 1
- 2 Timer/counter 2

Mode: The counter mode to set. Valid values are:

- 0 Event count
- 1 Programmable one shot
- 2 Rate generator
- 3 Square wave rate generator
- 4 Software triggered strobe
- 5 Hardware triggered strobe

Return Value:

true: Success.

false: Failure with errno set as follows:

EINVAL Timer is not valid.

EINVAL Mode is not valid.

Please see the description of the internal function outb() for information on other possible values errno may have in this case.

IOCTL Interface:

```
dm6812_ioctl_argument_t ioctl_request;
int file_descriptor;
int status;

/*
 * Before calling ioctl(), file_descriptor must be set up.  This is not shown
 * here.
 */

/*
 * Write to the 8254 Timer/Counter Control Word Register at base I/O address + 15
 */

ioctl_request.access_8.offset = 0x0F;

/*
 * Set timer to read/load least significant byte first then most significant
 * byte.  Because bit 0 is set to zero, this puts the timer in binary mode.
 */

ioctl_request.access_8.data = 0x30;

/*
 * Operate on timer/counter 1
 */

ioctl_request.access_8.offset |= 0x40;

/*
 * Put timer in rate generator mode
 */

ioctl_request.access_8.offset |= 0x04;
```

```
status = ioctl(file_descriptor, DM6812_IOCTL_WRITE_8_BITS, &ioctl_request);
```

CloseBoard6812

bool CloseBoard6812(void);

Description:

Close a DM6812 device file.

Parameters:

None.

Return Value:

true:	Success.
false:	Failure. Please see the close(2) man page for information on possible values errno may have in this case.

IOCTL Interface:

None.

DIOClearChip6812

bool DIOClearChip6812(u_int8_t Port);

Description:

Clear the digital I/O chip for the given digital I/O port.

NOTE: The digital I/O ports exist in pairs. Ports 0 & 1 form a pair, as do ports 2 & 3, and ports 4 & 5. Therefore, clearing the digital I/O chip for a port also clears the digital I/O chip for the other port in the pair.

Parameters:

Port: Digital I/O port for which to clear chip. Valid values are:

- 0 Digital I/O port 0
- 1 Digital I/O port 1
- 2 Digital I/O port 2
- 3 Digital I/O port 3
- 4 Digital I/O port 4
- 5 Digital I/O port 5

Return Value:

true: Success.

false: Failure with errno set as follows:

EINVAL Port is not valid.

Please see the descriptions of the internal functions outb() and select_dio_control_register() for information on other possible values errno may have in this case.

IOCTL Interface:

This function makes use of several ioctl() requests.

DIOClearStrobe6812

---

bool DIOClearStrobe6812(u_int8_t Port);

Description:

Clear the Strobe Status bit in the Read Digital I/O Status Register for the given digital I/O port.

NOTE: DIOClearChip6812() does not clear the Strobe Status bit. During program initialization, DIOClearStrobe6812() must be called to ensure that the bit is cleared. Once data is being strobed into an even-numbered port, reading data from that port's Compare Register will clear the Strobe Status bit.

Parameters:

Port: Digital I/O port to clear strobe status of. Valid values are:
- 0 Digital I/O port 0
- 2 Digital I/O port 2
- 4 Digital I/O port 4

Return Value:

true: Success.

false: Failure with errno set as follows:

EINVAL Port is not valid.

EOPNOTSUPP Port does not support data strobing.

Please see the descriptions of the internal functions inb() and select_dio_control_register() for information on other possible values errno may have in this case.

IOCTL Interface:

This function makes use of several ioctl() requests.

DIOEnableIrq6812

---

bool DIOEnableIrq6812(u_int8_t Port, bool Enable);

Description:

Enable or disable digital interrupts for the given digital I/O port.

NOTE: Only the even-numbered ports support digital interrupts.

NOTE: The DM6812 supports data strobe digital interrupts. In order to use these interrupts, you must disable digital interrupts on the port and set the port into match mode.

Parameters:

Port: Digital I/O port to modify interrupt state for. Valid values are:
- 0 Digital I/O port 0
- 2 Digital I/O port 2
- 4 Digital I/O port 4

Enable: Flag indicating whether or not digital interrupts should be enabled. A value of true means enable digital interrupts. A value of false means disable digital interrupts.

Return Value:

true: Success.

false: Failure with errno set as follows:

EINVAL Port is not valid.

EOPNOTSUPP Port does not support digital interrupts.

Please see the descriptions of the internal functions inb() and outb() for information on other possible values errno may have in this case.

IOCTL Interface:

This function makes use of several ioctl() requests.

bool DIOIsStrobe6812(uint8_t Port, bool *strobe_p);

Description:

Determine whether or not data has been strobed into the given digital I/O port.

NOTE: Only the even-numbered ports support data strobing.

Parameters:

Port: Digital I/O port to examine strobe status of. Valid values are:
- 0 Digital I/O port 0
- 2 Digital I/O port 2
- 4 Digital I/O port 4

strobe_p: Address where strobe flag should be stored. False will be stored here if no data strobe occurred. True will be stored here if a data strobe occurred. The contents of this memory is undefined if the function fails.

Return Value:

true: Success.

false: Failure with errno set as follows:

EINVAL Port is not valid.

EOPNOTSUPP Port does not support data strobing.

Please see the description of the internal function inb() for information on other possible values errno may have in this case.

IOCTL Interface:

```
dm6812_ioctl_argument_t ioctl_request;
int file_descriptor;
int status;

/*
 * Before calling ioctl(), file_descriptor must be set up.  This is not shown
 * here.
 */

/*
 * Read the Port 4/5 Digital I/O Status Register at base I/O address + 11
 */

ioctl_request.access_8.offset = 0x0B;
```

```
/*
 * This value does not matter because it is ignored making the request.  However
 * after ioctl() returns, the structure member will contain the register
 * contents.
 */

ioctl_request.access_8.data = 0;

status = ioctl(file_descriptor, DM6812_IOCTL_READ_8_BITS, &ioctl_request);
if (status == 0) {

    /*
     * If bit 7 is set, then data strobe occurred on port 4
     */

    if (ioctl_request.access_8.data & 0x80) {
        fprintf(stdout, "Data strobed into digital I/O port 4\n");
    }
}
```

DIORead6812

bool DIORead6812(uint8_t Port, uint8_t *data_p);

Description:

Read an 8-bit value from the given digital I/O port.

Parameters:

Port: The port to read from. Value values are:

0	Digital I/O port 0
1	Digital I/O port 1
2	Digital I/O port 2
3	Digital I/O port 3
4	Digital I/O port 4
5	Digital I/O port 5

data_p: Address where data read should be stored. The contents of this memory is undefined if the function fails.

Return Value:

true: Success.

false: Failure with errno set as follows:

EINVAL Port is not valid.

Please see the description of the internal function inb() for information on other possible values errno may have in this case.

IOCTL Interface:

```
dm6812_ioctl_argument_t ioctl_request;
int file_descriptor;
int status;

/*
 * Before calling ioctl(), file_descriptor must be set up.  This is not shown
 * here.
 */

/*
 * Read the Digital I/O Port 1 Register at base I/O address + 1
 */

ioctl_request.access_8.offset = 0x01;

/*
 * This value does not matter because it is ignored making the request.
 * However after ioctl() returns, the structure member will contain the
 * register contents.
 */

ioctl_request.access_8.data = 0;

status = ioctl(file_descriptor, DM6812_IOCTL_READ_8_BITS, &ioctl_request);
if (status == 0) {
    fprintf(stdout, "Port 1 input: 0x%x\n", ioctl_request.access_8.data);
}
```

DIOReadCompare6812

---

bool DIOReadCompare6812(uint8_t port, uint8_t *compare_p);

Description:

Read the Compare Register value for the given digital I/O port.

NOTE: Only the even-numbered ports have a Compare Register.

NOTE: When using data strobe and event interrupt modes, the port value which caused the interrupt is latched into the Compare Register and can be read from it.

Parameters:

port: Port to read Compare Register of. Valid values are:
- 0 Digital I/O port 0
- 2 Digital I/O port 2
- 4 Digital I/O port 4

compare_p: Address where register value will be stored. The contents of this memory is undefined if the function fails.

Return Value:

true: Success.

false: Failure with errno set as follows:

EINVAL port is not valid.

EOPNOTSUPP port does not support Compare Register.

Please see the descriptions of the internal functions inb() and select_dio_control_register() for information on other possible values errno may have in this case.

IOCTL Interface:

This function makes use of several ioctl() requests.

DIOSelectClock6812

---

bool DIOSelectClock6812(uint8_t Port, bool Programmable);

Description:

Select sampling clock for the given digital I/O port.

NOTE: The digital I/O ports exist in pairs. Ports 0 & 1 form a pair, as do ports 2 & 3, and ports 4 & 5. Therefore, setting the sampling clock for a port also sets the same sampling clock for the other port in the pair.

Parameters:

Port: Port to set clock for. Valid values are:

0 Digital I/O port 0
1 Digital I/O port 1
2 Digital I/O port 2
3 Digital I/O port 3
4 Digital I/O port 4
5 Digital I/O port 5

Programmable: Flag to indicate whether or not 8254 Timer/Counter 1 should be used. A value of false means use the 8 MHz system clock. A value of true mean use the 8254 Timer/Counter 1.

Return Value:

true: Success.

false: Failure with errno set as follows:

EINVAL Port is not valid.

Please see the descriptions of the internal functions inb() and outb() for information on other possible values errno may have in this case.

IOCTL Interface:

This function makes use of several ioctl() requests.

DIOSetBitDirection6812

---

bool DIOSetBitDirection6812(uint8_t Port, uint8_t Direction);

Description:

Set the direction (input or output) for the bits in the given digital I/O port.

NOTE: Only the even-numbered ports are bit direction programmable.

Parameters:

Port: Port to set bit direction for. Valid values are:
0 Digital I/O port 0
2 Digital I/O port 2
4 Digital I/O port 4

Direction: Bit mask which controls bit direction. A zero in a bit position means the corresponding port bit is set to input. A one in a bit position means the corresponding port bit is set to output.

Return Value:

true: Success.

false: Failure with errno set as follows:

EINVAL Port is not valid.

EOPNOTSUPP Port is not bit direction programmable.

Please see the descriptions of the internal functions outb() and select_dio_control_register() for information on other possible values errno may have in this case.

IOCTL Interface:

This function makes use of several ioctl() requests.

DIOSetCompareValue6812

---

bool DIOSetCompareValue6812(uint8_t Port, uint8_t Value);

Description:

Set the Compare Register value for the given digital I/O port.

NOTE: Only the even-numbered ports have a Compare Register.

NOTE: A port's Compare Register can be written to only if the port is set to match mode and digital interrupts are enabled for the port.

Parameters:

Port: Port to set Compare Register for. Valid values are:
- 0 Digital I/O port 0
- 2 Digital I/O port 2
- 4 Digital I/O port 4

Value: Value to store in Compare Register.

Return Value:

true: Success.

false: Failure with errno set as follows:

EINVAL Port is not valid.

EOPNOTSUPP Port does not support Compare Register.

Please see the descriptions of the internal functions outb() and select_dio_control_register() for information on other possible values errno may have in this case.

IOCTL Interface:

This function makes use of several ioctl() requests.

DIOSetIrqMode6812

bool DIOSetIrqMode6812(uint8_t Port, bool MatchMode);

Description:

Set the digital IRQ mode for the given digital I/O port.

NOTE: Only the even-numbered ports support digital interrupts.

Parameters:

Port: Port to set IRQ mode for. Valid values are:
- 0 Digital I/O port 0
- 2 Digital I/O port 2
- 4 Digital I/O port 4

MatchMode: Flag indicating whether or not match mode interrupt should be enabled. A value of false means enable event mode interrrupts. A value of true means enable match mode interrupts.

Return Value:

true: Success.

false: Failure with errno set as follows:

EINVAL Port is not valid.

EOPNOTSUPP Port does not support digital interrupts.

Please see the descriptions of the internal functions inb() and outb() for information on other possible values errno may have in this case.

IOCTL Interface:

This function makes use of several ioctl() requests.

DIOSetMaskValue6812

bool DIOSetMaskValue6812(uint8_t Port, uint8_t Mask);

Description:

Set the Mask Register value for the given digital I/O port.

NOTE: Only the even-numbered ports have a Mask Register.

Parameters:

Port: Port to set Compare Register for. Valid values are:
- 0 Digital I/O port 0
- 2 Digital I/O port 2
- 4 Digital I/O port 4

Mask: Value to store in Mask Register. When in event or match mode, a zero in a bit position means that the corresponding port bit can generate a digital interrupt. When not using digital interrupts, a zero in a bit position means that the corresponding port bit can change state if being used for output. When in event or match mode, a one in a bit position means that the corresponding port bit cannot generate a digital interrupt. When not using digital interrupts, a one in a bit position means that the corresponding port bit cannot change state if being used for output.

Return Value:

true: Success.

false: Failure with errno set as follows:

EINVAL Port is not valid.

EOPNOTSUPP Port does not support Mask Register.

Please see the descriptions of the internal functions outb() and select_dio_control_register() for information on other possible values errno may have in this case.

IOCTL Interface:

This function makes use of several ioctl() requests.

DIOSetPortDirection6812

---

bool DIOSetPortDirection6812(uint8_t Port, bool Output);

Description:

Set the direction (input or output) for all bits in the given digital I/O port.

NOTE: Only the odd-numbered ports are byte direction programmable.

Parameters:

Port: Port to set bit direction for. Valid values are:
- 1 Digital I/O port 1
- 3 Digital I/O port 3
- 5 Digital I/O port 5

Output: Flag indicating whether or not all port bits should be set to output. A value of true means set all bits to output. A value of false means set all bits to input.

Return Value:

true: Success.

false: Failure with errno set as follows:

EINVAL Port is not valid.

EOPNOTSUPP Port is not byte direction programmable.

Please see the descriptions of the internal functions inb() and outb() for information on other possible values errno may have in this case.

IOCTL Interface:

This function makes use of several ioctl() requests.

DIOWrite6812

---

bool DIOWrite6812(uint8_t Port, uint8_t Data);

Description:

Write an 8-bit value to the given digital I/O port.

Parameters:

Port: The port to write to. Value values are:
- 0 Digital I/O port 0
- 1 Digital I/O port 1
- 2 Digital I/O port 2
- 3 Digital I/O port 3
- 4 Digital I/O port 4
- 5 Digital I/O port 5

Data: Data to write. Valid values are 0 through 255.

Return Value:

true: Success.

false: Failure with errno set as follows:

EINVAL Port is not valid.

Please see the description of the internal function outb() for information on other possible values errno may have in this case.

IOCTL Interface:

```
dm6812_ioctl_argument_t ioctl_request;
int file_descriptor;
int status;

/*
 * Before calling ioctl(), file_descriptor must be set up.  This is not shown
 * here.
 */

/*
 * Write to the Digital I/O Port 3 Register at base I/O address + 5
 */

ioctl_request.access_8.offset = 0x05;

/*
 * Write a value with bits 4 through 7 set to one
 */

ioctl_request.access_8.data = 0xF0;

status = ioctl(file_descriptor, DM6812_IOCTL_WRITE_8_BITS, &ioctl_request);
```

DM6812Device

DM6812Device(void);

Description:

DM6812Device class constructor.

Parameters:

None.

Return Value:

None. Constructors do not return a value.

IOCTL Interface:

None.

~DM6812Device

~DM6812Device(void);

Description:

DM6812Device class destructor.

NOTE: This function closes the DM6812 device file associated with the object.

Parameters:

None.

Return Value:

None. Destructors do not return a value.

IOCTL Interface:

None.

GetDriverVersion6812

bool GetDriverVersion6812(uint32_t *version_p);

Description:

Get the driver version number. The version number is an unsigned integer encoding the major, minor, and patch level numbers.

NOTE: The driver version is encoded according to the formula

```
Version = (
    (MajorVersion << 16)
    |
    (MinorVersion << 8)
    |
    PatchLevelNumber
)
```

Parameters:

version_p: Address where version number should be stored. The contents of this memory is undefined if the function fails.

Return Value:

true: Success.

false: Failure. Please see the ioctl(2) man page for information on possible values errno may have in this case.

IOCTL Interface:

```
dm6812_ioctl_argument_t ioctl_request;
int file_descriptor;
int status;

/*
 * Before calling ioctl(), file_descriptor must be set up.  This is not shown
 * here.
 */

status = ioctl(file_descriptor, DM6812_IOCTL_GET_DRIVER_VERSION, &ioctl_request);
if (status == 0) {
    u_int32_t  version;

    version = ioctl_request.version.driver_version;

    fprintf(stdout, "Major version: %d\n", ((version >> 16) & 0xF));
    fprintf(stdout, "Minor version: %d\n", ((version >> 8) & 0xF));
    fprintf(stdout, "Patch level: %d\n", (version & 0xF));
}
```

GetIntStatus6812

bool GetIntStatus6812(uint32_t *int_count_p, uint8_t *status_reg_p);

Description:

Atomically obtain the driver's current interrupt count and cached IRQ Status Register value.

NOTE: This function clears the driver's cached IRQ Status Register value. The value will remain cleared until the next interrupt occurs.

NOTE: Only bits 0 through 3 in the cached IRQ Status Register value are returned.

NOTE: The macros P14_INT_OCCURRED(), PORT4_INT_OCCURRED(), PORT2_INT_OCCURRED(), and PORT0_INT_OCCURRED() should be used to examine specific Status Register value bits to determine the type(s) of interrupt(s) which occurred. Each macro returns true if the associated interrupt occurred and false if it did not.

Parameters:

int_count_p:	Address where interrupt count should be stored. The contents of this memory is undefined if the function fails.
status_reg_p:	Address where Status Register value should be stored. The contents of this memory is undefined if the function fails.

Return Value:

true:	Success.
false:	Failure. Please see the ioctl(2) man page for information on possible values errno may have in this case.

IOCTL Interface:

```
dm6812_ioctl_argument_t ioctl_request;
int file_descriptor;
int status;

/*
 * Before calling ioctl(), file_descriptor must be set up.  This is not shown
 * here.
 */

/*
 * These values do not matter because they are ignored making the request.
 * However after ioctl() returns, the structure members will contain the
 * requested information.
 */

ioctl_request.int_status.status_reg = 0;
ioctl_request.int_status.int_count = 0;

status = ioctl(file_descriptor, DM6812_IOCTL_GET_INT_STATUS, &ioctl_request);
if (status != 0) {
    exit(EXIT_FAILURE);
}

if (PORT0_INT_OCCURRED(ioctl_request.int_status.status_reg)) {
    fprintf(
        stdout,
        "Port 0 interrupt occurred, count = %u\n",
        ioctl_request.int_status.int_count
    );
}
```

LoadIRQRegister6812

---

bool LoadIRQRegister6812(uint8_t Value);

Description:

Load an 8-bit value into a board's Clear IRQ/IRQ Enable Register at base I/O address + 16.

NOTE: The DM6812 interrupt handler is not designed to process interrupts shared between devices. To avoid unpredictable behavior or worse, do not share an interrupt between devices.

NOTE: Interrupts do not need to be shared in order to use several interrupt sources on a single board. For example if you wish to use both P14 and port 0 digital interrupts on a single DM6812 device, then interrupts do not need to be shared.

NOTE: The macros ENABLE_IRQ_SHARING(), DISABLE_IRQ_SHARING(), POSITIVE_IRQ_POLARITY(), NEGATIVE_IRQ_POLARITY(), ENABLE_P14_IRQ(), and DISABLE_P14_IRQ() should be used to set or clear bits in the value passed to this function.

Parameters:

Value: Value to store in register. Valid values are 0 through 7. Please see the hardware manual for the interpretation of the register's bits.

Return Value:

true:	Success.
false:	Failure. Please see the description of the internal function outb() for information on possible values errno may have in this case.

IOCTL Interface:

```
dm6812_ioctl_argument_t ioctl_request;
int file_descriptor;
int status;

/*
 * Before calling ioctl(), file_descriptor must be set up.  This is not shown
 * here.
 */

/*
 * Write to the IRQ Enable Register at base I/O address + 16
 */

ioctl_request.access_8.offset = 0x10;

/*
 * Set up P14 interrupt
 */

ioctl_request.access_8.data = 0x00;
DISABLE_IRQ_SHARING(ioctl_request.access_8.data);
NEGATIVE_IRQ_POLARITY(ioctl_request.access_8.data);
ENABLE_P14_IRQ(ioctl_request.access_8.data);

status = ioctl(file_descriptor, DM6812_IOCTL_WRITE_8_BITS, &ioctl_request);
```

OpenBoard6812

bool OpenBoard6812(uint32_t nDevice);

Description:

Open a DM6812 device file.

Parameters:

nDevice:	Minor number of DM6812 device file.

Return Value:

true:	Success.
false:	Failure. Please see the open(2) man page for information on possible values errno may have in this case.

IOCTL Interface:

None.

bool ReadByte6812(uint8_t offset, uint8_t *data_p);

Description:

Read an 8-bit value from the given offset within a DM6812 board's I/O memory.

NOTE: It is strongly suggested that you use other library functions instead of directly accessing a board's registers.

Parameters:

offset: Offset within I/O memory to read.

data_p: Address where data read should be stored. The contents of this memory is undefined if the function fails.

Return Value:

true: Success.

false: Failure with errno set as follows:

EINVAL offset is not valid.

EOPNOTSUPP offset is valid but it represents a write-only register.

Please see the ioctl(2) man page for information on other possible values errno may have in this case.

IOCTL Interface:

```
dm6812_ioctl_argument_t ioctl_request;
int file_descriptor;
int status;

/*
 * Before calling ioctl(), file_descriptor must be set up.  This is not shown
 * here.
 */

/*
 * Read the IRQ Status Register at base I/O address + 17
 */

ioctl_request.access_8.offset = 0x11;

/*
 * This value does not matter because it is ignored making the request.
 * However after ioctl() returns, the structure member will contain the
 * register contents.
 */

ioctl_request.access_8.data = 0;

status = ioctl(file_descriptor, DM6812_IOCTL_READ_8_BITS, &ioctl_request);
```

ReadTimerCounter6812

bool ReadTimerCounter6812(uint8_t Timer, uint16_t *count_p);

Description:

Read the count for the given timer/counter.

Parameters:

Timer: The timer to operate on. Valid values are:
- 0 Timer/counter 0
- 1 Timer/counter 1
- 2 Timer/counter 2

count_p: Address where timer count should be stored. The contents of this memory is undefined if the function fails.

Return Value:

true: Success.

false: Failure with errno set as follows:

EINVAL Timer is not valid.

Please see the descriptions of the internal functions outb() and inb() for information on other possible values errno may have in this case.

IOCTL Interface:

This function makes use of several ioctl() requests.

SetUserClock6812

bool SetUserClock6812(uint8_t Timer, float InputRate, float OutputRate, float *actual_rate_p);

Description:

Set the given timer/counter into rate generator mode and program its divisor value based upon the specified input and output rates.

Parameters:

Timer: The timer to program. Valid values are:
- 0 Timer/counter 0
- 1 Timer/counter 1
- 2 Timer/counter 2

InputRate: Input clock rate to timer/counter.

OutputRate: Desired output rate.

actual_rate_p: Address where actual programmed frequency should be stored. The contents of this memory is undefined if the function fails.

Return Value:

true: Success.

false: Failure. Please see the descriptions of ClockDivisor6812() and ClockMode6812() for information on possible values errno may have in this case.

IOCTL Interface:

This function makes use of several ioctl() requests.

WaitForInterrupt6812

---

bool WaitForInterrupt6812(void);

Description:

Wait for an interrupt to occur on a device.

NOTE: Signals can wake up the process before an interrupt occurs. If a signal is delivered to the process during a wait, the application is responsible for dealing with the premature awakening in a reasonable manner.

NOTE: Because this function can be woken up by a signal before an interrupt occurs, an interrupt may be missed if signals are delivered rapidly enough or at inopportune times. To decrease the chances of this, it is strongly suggested that you 1) do not use signals or 2) minimize their use in your application.

Parameters:

None.

Return Value:

true: Success.

false: Failure with errno set as follows:

EINTR The process received a signal before an interrupt occurred. This is not a fatal error but rather means the wait should be retried.

ENODATA No signal was delivered and select() woke up without indicating that an interrupt occurred. This indicates serious problems within the driver.

Please see the select(2) man page for information on other possible values errno may have in this case.

IOCTL Interface:

None.

WriteByte6812

---

bool WriteByte6812(uint8_t offset, uint8_t data);

Description:

Write an 8-bit value to the given offset within a DM6812 board's I/O memory.

NOTE: It is strongly suggested that you use other library functions instead of directly accessing a board's registers.

Parameters:

offset: Offset within I/O memory to write.

data: Data to write.

Return Value:

true: Success.

false: Failure with errno set as follows:

EINVAL offset is not valid.

EOPNOTSUPP offset is valid but it represents a read-only register.

Please see the ioctl(2) man page for information on other possible values errno may have in this case.

IOCTL Interface:

```
dm6812_ioctl_argument_t ioctl_request;
int file_descriptor;
int status;

/*
 * Before calling ioctl(), file_descriptor must be set up.  This is not shown
 * here.
 */

/*
 * Write to the Digital I/O Port 1 Register at base I/O address + 1
 */

ioctl_request.access_8.offset = 0x01;

/*
 * Write all zeros to the port
 */

ioctl_request.access_8.data = 0x00;

status = ioctl(file_descriptor, DM6812_IOCTL_WRITE_8_BITS, &ioctl_request);
```

Example Programs Reference

Name	Remarks
basic-test	Tests the basic functionality of the driver and library.
digital-io	Demonstrates reading from and writing to the digital I/O ports.
dio-test	Tests the library functions related to digital I/O.
event-int	Demonstrates how to use event mode digital interrupts.
external-int	Demonstrates how to use P14 external interrupts.
interrupt-test	Tests the library interrupt-related functions.
interrupt-wait	Demonstrates waiting for interrupt notification.
match-int	Demonstrates how to use match mode digital interrupts.
strobe-int	Demonstrates how to use data strobe mode digital interrupts.
timer-int	Demonstrates how to use P14 timer interrupts.
timers	Demonstrates using the 8254 timer/counters in rate generator and event count modes.
timer-test	Tests the basic functionality of the 8254 timer/counter library functions.

Limited Warranty

RTD Embedded Technologies, Inc. warrants the hardware and software products it manufactures and produces to be free from defects in materials and workmanship for one year following the date of shipment from RTD Embedded Technologies, INC. This warranty is limited to the original purchaser of product and is not transferable.

During the one year warranty period, RTD Embedded Technologies will repair or replace, at its option, any defective products or parts at no additional charge, provided that the product is returned, shipping prepaid, to RTD Embedded Technologies. All replaced parts and products become the property of RTD Embedded Technologies. Before returning any product for repair, customers are required to contact the factory for an RMA number.

THIS LIMITED WARRANTY DOES NOT EXTEND TO ANY PRODUCTS WHICH HAVE BEEN DAMAGED AS A RESULT OF ACCIDENT, MISUSE, ABUSE (such as: use of incorrect input voltages, improper or insufficient ventilation, failure to follow the operating instructions that are provided by RTD Embedded Technologies, "acts of God" or other contingencies beyond the control of RTD Embedded Technologies), OR AS A RESULT OF SERVICE OR MODIFICATION BY ANYONE OTHER THAN RTD Embedded Technologies. EXCEPT AS EXPRESSLY SET FORTH ABOVE, NO OTHER WARRANTIES ARE EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, ANY IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE, AND RTD Embedded Technologies EXPRESSLY DISCLAIMS ALL WARRANTIES NOT STATED HEREIN. ALL IMPLIED WARRANTIES, INCLUDING IMPLIED WARRANTIES FOR MECHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE, ARE LIMITED TO THE DURATION OF THIS WARRANTY. IN THE EVENT THE PRODUCT IS NOT FREE FROM DEFECTS AS WARRANTED ABOVE, THE PURCHASER'S SOLE REMEDY SHALL BE REPAIR OR REPLACEMENT AS PROVIDED ABOVE. UNDER NO CIRCUMSTANCES WILL RTD Embedded Technologies BE LIABLE TO THE PURCHASER OR ANY USER FOR ANY DAMAGES, INCLUDING ANY INCIDENTAL OR CONSEQUENTIAL DAMAGES, EXPENSES, LOST PROFITS, LOST SAVINGS, OR OTHER DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE PRODUCT.

SOME STATES DO NOT ALLOW THE EXCLUSION OR LIMITATION OF INCIDENTAL OR CONSEQUENTIAL DAMAGES FOR CONSUMER PRODUCTS, AND SOME STATES DO NOT ALLOW LIMITATIONS ON HOW LONG AN IMPLIED WARRANTY LASTS, SO THE ABOVE LIMITATIONS OR EXCLUSIONS MAY NOT APPLY TO YOU.

THIS WARRANTY GIVES YOU SPECIFIC LEGAL RIGHTS, AND YOU MAY ALSO HAVE OTHER RIGHTS WHICH VARY FROM STATE TO STATE.

RTD Embedded Technologies, Inc.
103 Innovation Boulevard
State College PA 16803-0906
USA
Our website: www.rtd.com